A Visual Network Analysis Method for Large Scale Parallel I/O Systems

Carmen Sigovan*, Chris Muelder*, Kwan-Liu Ma*
*University of California Davis
{cmsigovan, cwmuelder, klma}@ucdavis.edu

Jason Cope†, Kamil Iskra†, Robert Ross†
†Argonne National Laboratory,
Mathematics and Computer Science Division
{copej, iskra, rross}@mcs.anl.gov

***Abstract*—Parallel applications rely on I/O to load data, store end results and to protect partial results from being lost to system failure. Parallel I/O performance thus has a direct and significant impact on application performance. Because supercomputer I/O systems are very large and complex, it is impossible to directly analyze their activity traces. While several visual or automated analysis tools for large scale HPC log data exist, analysis research in the high performance computing field is geared toward computation performance rather than toward I/O performance. Additionally, existing methods usually do not capture the network characteristics of HPC I/O systems. We present a visual analysis method for I/O trace data that takes into account the fact that HPC I/O system can be represented as a network. We illustrate performance metrics in a way that facilitates the identification of abnormal behavior or performance problems. We demonstrate our approach on I/O traces collected from existing systems at different scales.**

I. Introduction

Modern HPC system design typically separates compute resources from on-line storage resources, for a number of reasons, including improved power utilization and reliability of compute resources and ease of storage management. However, this configuration also introduces the necessity for additional hardware and software layers to manage the transfer of data between the compute cluster and permanent storage [1], as shown in Figure 1. These layers provide tools for parallel application developers to effectively utilize the available HPC resources, while not requiring them to be aware of the inner workings of the I/O system.

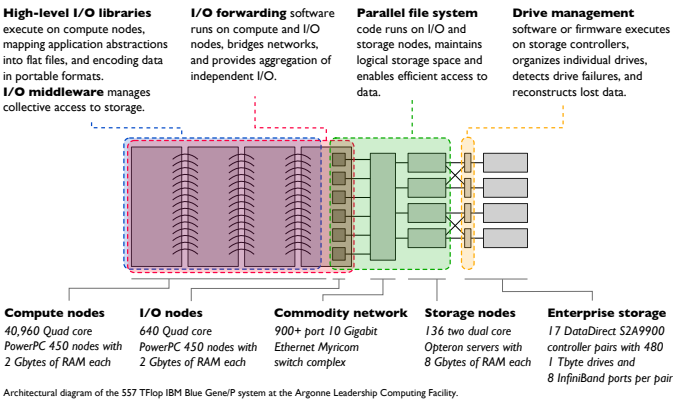


Fig. 1. I/O system configuration of the IBM Blue Gene/P supercomputer. The compute nodes and I/O nodes are part of the same network and the storage nodes are connected to this network by a separate Ethernet network.

Conversely, the additional layers also increase the overall complexity of the system, thus making I/O performance analysis more difficult. Because the vast majority of parallel applications rely on the HPC I/O stack for their data management operations, it is important that we understand the factors which influence I/O performance. This understanding will allow us to better detect and correct problems and to possibly enhance system performance via the implementation of new solutions.

It is expected that I/O performance will continue to be a significant bottleneck as the size and power of HPC systems increase [2]. Understanding the complex interactions between the software and hardware layers of the HPC I/O stack is a key prerequisite for I/O performance analysis and improvement. Unfortunately, the complex, layered nature of I/O systems makes analysis nearly impossible without the use of specialized tools. The sheer number of nodes and the events they generate result in data sets that are far too large for direct analysis methods. Small tests generate hundreds of thousands of events, with medium to large tests potentially resulting in millions of events per second of execution and taking up gigabytes of space when written to log files. This growth in system and in trace data scale is the motivation for creating novel visualization techniques, which can provide useful information about the state and performance of the system at a glance.

Since typical HPC I/O subsystems consist of sets of interconnected components, network performance characteristics are quite relevant to the performance of the I/O system. We therefore designed a procedure of extracting network performance metrics from I/O trace data and a visualization method by which we illustrate these metrics as they relate to the activity of the I/O system. We record trace data from three layers of the system: the application layer (MPI-IO and POSIX I/O calls), the I/O nodes (responsible for transmitting I/O requests to the storage cluster) and from the storage nodes (which have direct access to the storage hardware). The resulting trace data is a record of all requests and communication events that have occurred over the course of an application or benchmark execution. We process this data to extract network performance metrics, such as latency and throughput, and build a complete view of the I/O system's behavior.

To summarize, we have devised a general method for I/O system performance analysis that would be applicable to I/O trace data collected from any type of HPC system. We have also performed a series of case studies of I/O benchmark kernels on the IBM Blue Gene/P to demonstrate the capabilities of our method.

II. Related Work

Parallel I/O performance has long been a topic of interest for the research community. Consequently, there are numerous studies of I/O performance for large-scale parallel systems. Many such studies differ from ours in that they utilize overall performance metrics from multiple application runs at different scales to characterize I/O system performance. Often, when a new type of supercomputer becomes available, HPC researchers perform tests to determine its exact performance and behavior under certain loads [3], [4]. Other research studies focus on understanding parallel I/O system performance as it relates to a particular class of applications, such as parallel visualization applications [5], [6], or large-scale scientific simulations [7]. Furthermore, there are studies that are only focused on analyzing the I/O performance of a particular parallel program [8]. In our work, we have endeavored to develop a portable I/O tracing system and visualization methods that are general enough to be useful in the analysis of any I/O trace, while still being specifically designed for the exploration of parallel I/O performance data.

There are two main areas of related work closely pertaining to our project. Our work is based both on research of data collection techniques from parallel systems and on visualization techniques for parallel and network data.

A. Data Collection

HPC software instrumentation and tracing are active areas of research with a wide breadth of research topics. In our work, we adopt successful techniques rather than building new ones, and we focus our efforts on gaps in existing tools.

Some mainstream tracing solutions would not be a good fit for our purpose because they require software or hardware configurations not available on the systems we are working with. LANL-Trace [9], for instance, relies on general-purpose compute-node kernels and dynamically linked libraries, which are not available on IBM Blue Gene systems using the lightweight CNK kernel. HPCT-I/O [10] and IOT [7] are two examples of I/O tracing toolkits developed specifically for leadership class architectures, respectively IBM Blue Gene and Cray XT. However, the results published so far have all been performed at small scale, so it is too early to say how these toolkits will function at HPC scales. Recently, the scalability of Scalasca was improved up to $300,000$ cores [11]. TAU [12] is a flexible program and performance analysis toolkit that supports parallel tracing and has a field-proven scaling record, having been used at full scale on IBM Blue Gene, Cray XT, and others. It is a generic tool framework that can be used for a variety of performance analysis tasks, I/O tracing included.

One successful example of generating large-scale I/O traces is the work of the Sandia Scalable I/O team, which released the traces of several parallel applications ran at a scale of 2744–6400 processes on Red Storm, a Cray XT3-class machine [13]. The traces were obtained by incorporating a lightweight tracing capability [14] into the SYSIO library [15], a user-level VFS layer linked into the applications on that platform.

More recently, researchers have proposed automated tracing and instrumentation tools for parallel I/O [16]. Such approaches combine code analysis with tracing instrumentation to automatically extract I/O performance data from parallel applications.

B. Data Analysis and Visualization

When confronted with relatively large amounts of parallel event data, the use of automated analysis (machine learning) or visualization methods becomes a necessity if we are to extract useful information out of such data sets. This necessity has driven the development of several analysis tools for parallel data. For instance, KOJAK (Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks) [17]–[19] and Scalasca [20] both perform automated analysis on parallel execution traces and detect performance issues.

Visual methods are quite effective when the analyst prefers more direct access to the data, but they can also be coupled with automatic pattern analysis to produce powerful results [21]. However, many well-known visual analysis tools for parallel trace data are focused on distilling information from MPI traces. Jumpshot [22] and its predecessors (Nupshot and Upshot [23], [24]) use the MPE library [25] to trace MPI calls in parallel applications and display the collected information in Gantt charts, using color to represent types of MPI calls and arrows to indicate communication between processes. Vampir [26] combines Gantt charts with aggregate views of system activity. Jumpshot and Vampir have both been successfully employed in the analysis of parallel I/O data as well. The TAU (Tuning and Analysis Utilities) [12] suite is a complete set of analysis tools for parallel performance data, including both tracing and visualization capabilities. TAU's visualization tools include Gantt charts, call graph views and communication matrices. Virtue is a unique parallel trace data visualization tool in that it employs virtual reality techniques to create an immersive and collaborative environment in which developers can interact with their parallel application in real time and can potentially adjust its behavior as it runs.

Because per-process activity representations such as Gantt charts tend to experience scalability issues, some visualization designs emphasize parallel event patterns and regard event to process association as secondary [27], [28]. This allows for a more scalable representation of large parallel traces. IOVis is a direct precursor to this work because it targets HPC I/O traces in particular and because it contains a matrix representation of communication patterns within the I/O system.

As we have seen, several analysis methods for parallel systems exist; generally, analyzing parallel trace data requires either machine learning methods, which automatically detect

areas of interest, or visual methods. Automated analysis is sometimes used in conjunction with visualization, resulting in very powerful analysis tools. However, more analysis tools are dedicated to the study of inter-process communication (MPI traces) than to parallel I/O. There are some instances of tools originally designed for MPI performance analysis being adapted for the study of I/O data as well [29], [30]. But there are still relatively few dedicated tools for I/O system analysis and it is exactly this gap that we are trying to fill.

III. Data Collection

The typical HPC I/O software stack consists of multiple layers of software that provide a variety of I/O capabilities for specific application I/O patterns, system software configurations, and system architectures, as was shown earlier in Figure 1.

Across the majority of HPC systems, applications store data on high-performance parallel file systems. These file systems include PVFS2 [31], Lustre [32], and GPFS [33]. A file system server processes application I/O requests through the file system client interface. The computation resource may run file system clients on all of its nodes or on a subset of the nodes in conjunction with I/O aggregation and forwarding tools. Examples of I/O forwarding tools include IOFSL [34], [35], ZOID [36], IBM's CIOD, and Cray's Data Virtualization Service.

At the application level, there are several application I/O interfaces. File system I/O interfaces, such as POSIX, provide direct access to the file system or I/O forwarding layer. MPI-IO provides a parallel I/O interface built on top of the file system's APIs; it coordinates and optimizes parallel I/O patterns. High-level and scientific data libraries provide mechanisms to generate self-describing and portable data sets.

The overall goal for these software layers is to provide the best possible I/O performance for common HPC application I/O patterns. Achieving this goal is often a difficult task for application developers because the cost of the high-level I/O operations in the lower layers of the I/O software stack is unknown. Additional information and insight about how these layers interact and what the cost of high-level operations is in subsequent layers will help isolate bottlenecks within the I/O software and identify areas of improvement for software developers.

To capture the end-to-end behavior of I/O requests, we integrated several instrumentation layers into the application, PVFS2 file system client and server, and (optionally) I/O forwarding. These data collection layers track the beginning and completion of file system I/O events and collect information about each event, such as event type and payload size. Each data collection layer collects information for the events initiated at that layer and adds identifiers to events to track the operation execution through underlying software layers.

At the application layer, we instrumented MPI-IO and POSIX calls, capturing information such as the start and end time, file identifier, and data payload size. We instrumented the IOFSL I/O forwarding infrastructure to capture request caching, merging, and other transformations performed while requests are forwarded from the compute nodes to the I/O nodes. Note that when using IBM's ciod forwarding, which replays every forwarded call one-by-one, no instrumentation is necessary, as the forwarding is completely transparent; hence, we skip the I/O forwarding layer from the graphs shown later in the paper. We instrumented PVFS2 client to report the communication between the users of the file system and the storage servers. PVFS2 daemons running on the storage servers are instrumented to track network communication and storage management operations.

We have deployed the tracing infrastructure on the systems at the Argonne Leadership Computing Facility (ALCF). We use the 40-rack Intrepid Blue Gene/P platform for generating application traces and the 100-node Eureka Linux cluster for generating PVFS2 server traces. Each BG/P rack contains 1024 compute nodes and 16 I/O nodes. Each compute node has a quad-core 850 MHz IBM PowerPC 450 processor and 2 GB of RAM. Each Eureka node has two quad-core Intel Xeon processors, 32 GB of RAM, and 230 GB of local scratch storage. Eureka and Intrepid share a 10 Gbps communication network. In addition to Intrepid, for smaller experiments we can also use Surveyor, a single-rack BG/P test and development system with identical internal architecture.

When tracing applications in the ALCF environment, we set up a temporary PVFS2 storage cluster on Eureka and mounted this file system on the allocated Intrepid I/O nodes. With this deployment, we have successfully traced applications utilizing up to 16,384 processes on Intrepid and up to 32 PVFS2 I/O servers on Eureka. The applications we have evaluated in this environment include the mpi-tile-io benchmark [37], the IOR benchmark [38], the FLASH I/O kernel [39], and the Chombo I/O kernel.

IV. Visualization Methodology

From our previous experiments with the I/O trace data, we discovered that visualizing the duration of communication events did not reveal particularly useful information regarding the state of the system. At the I/O node and storage node levels in particular, the Send and Receive events we record tend to have roughly the same duration, with slight variations linked to the amount of data being transferred. This is to be expected of a well-tuned system and is therefore not a valuable result. With this in mind, we decided to use metrics such as latency and transferred data size in our I/O network visualization. This has required us to essentially "read between the lines" of our trace data to visualize the I/O network rather than individual events. We use the times between certain events and their associated buffer sizes to construct our metrics, rather than relying on event duration alone.

Our approach involves adapting a hierarchical graph visualization method [40] for the analysis of large-scale parallel I/O data. Typical high performance I/O systems have a hierarchy of compute nodes, I/O nodes and storage nodes, which are connected by one or multiple networks. Our tracing system records the data transfers occurring through the I/O system

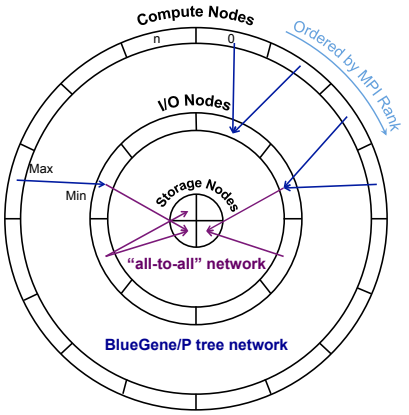


Fig. 2. Diagram of the graph display design.

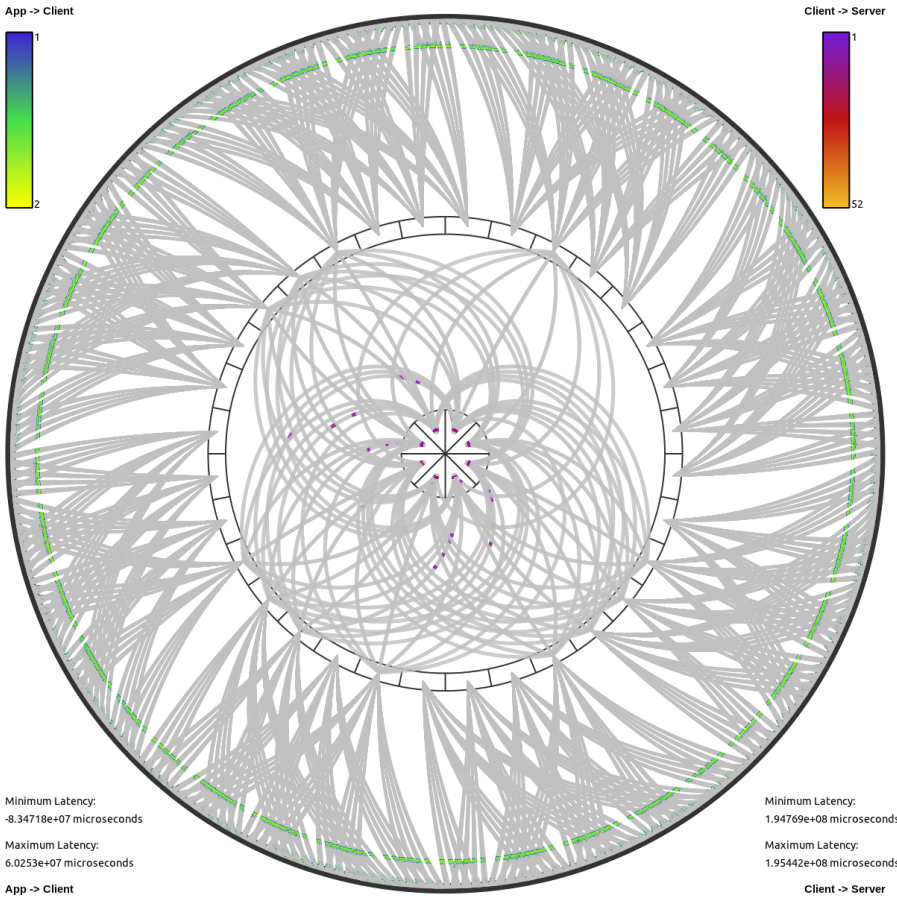


Fig. 3. I/O activity graph, depicting latency values measured on each link. This I/O trace contains 8192 compute processes, 32 I/O nodes and 8 storage nodes. The two types of network configuration (tree network and "all-to-all" Ethernet) are clearly visible between the layers. One each edge, colored segments indicate the latency values measured on that particular network link. In each layer of edges, colored segments closer to the edge of the circle (towards the outside) represent high values, while colored segments closer to the circle center indicate the lowest latency values found in the data. Color represents the number of occurrences of a particular latency value and will be explained in more detail in the upcoming paragraphs.

over the course of a parallel application's execution. Thus, we felt that this graph visualization method would be effective for our purposes. We modified the hierarchy representation to display the layers as concentric rings, with edges running between them depicting communication. Thanks to our hierarchical layout, our edge routing does not require bundling, thus saving computation time.

First, we construct the network hierarchy by scanning through the I/O trace and and building lists of all the compute nodes, I/O nodes and storage nodes discovered in the data. We then lay out the nodes in three concentric rings, one for each layer. The ordering of the rings is as follows: the outer ring contains the compute nodes, sorted by MPI rank. The middle ring consists of all the client nodes, sorted by the ordering in which the compute nodes communicate with them, and the innermost circle houses the server nodes, simply ordered by their node ID. We have chosen to lay out the nodes in this way because in our traces, there are always significantly more compute nodes than I/O nodes and more I/O nodes than storage nodes. Thus, we allocate more space to represent nodes in the denser layers. An overview of the design can be seen in Figure 2.

To construct the edges, we require the user to select a period of time from a timeline histogram. This period may be the entire duration of recorded execution or simply a fraction of time in the trace data. We then query the data for all the I/O events which occurred during the selected period of time. If the resulting events indicate that communication occurred between two nodes, then there is an edge between these nodes. For example, if we find a ClientSend event at the I/O node level, we look for a corresponding ServerReceive event at the storage node level to establish if there is communication between the I/O node and the storage node. Figure 3 shows an observed I/O communication graph.

A. Nodes and Edge Routing

We simplify our edge routing procedure by initially ordering the node representations at each layer in an advantageous

manner. The nodes from each layer are represented by ring segments or by circle sectors in the case of the I/O servers. On the ring corresponding to the application layer, processes are sorted ascendantly according to their MPI rank. We attempt to place client nodes on their ring according to which processes they receive requests from. In our case, the compute nodes and the I/O nodes are connected by the Blue Gene/P tree network, so any compute node may only send requests to one I/O node, while I/O nodes may receive requests from a number of compute nodes. Thus, our ordering of process IDs and I/O nodes in the hierarchy serves to minimize the number of edge crossings at this level. Edge crossings are, however, unavoidable when visualizing larger data sets containing thousands of compute nodes, such as the one depicted in Figure 3. The innermost circle represents the storage nodes, arranged in the order in which the nodes were detected in the data. Since the network between the I/O and the storage nodes has all-to-all connections, there is no particular ordering that we can apply on the storage nodes' layer.

Once the edge lists are generated, we render the graph, using linear interpolation of angles around the circle to give edges their curvature and to route them such that they do not cross the intermediate circles belonging to other layers unless they have reached their target node. For an edge with a start point ($startR$, $start\theta$) and an end point ($endR$, $end\theta$) in polar coordinates, intermediate point positions are calculated as follows:

$$inter\theta = start\theta + (end\theta - start\theta) \times interStep$$
$$interR = startR + (endR - startR) \times interStep$$

where *interStep* is an intermediate step in the interpolation defined as an integer between 0 and the total number of interpolation steps. We define the number of interpolation steps such that the edges appear as curves and no interpolation artifacts are visible. We also route edges on the shortest path around the circle by disallowing the difference between the end and start angles to exceed π or to be smaller than $-\pi$. We adjust the angles as follows:

$$end\theta = \begin{cases} end\theta - 2\pi, & \text{if } (end\theta - start\theta) \geq \pi \\ end\theta + 2\pi, & \text{if } (end\theta - start\theta) < \pi \end{cases}$$

B. Performance Metrics and the Color Map

In addition to revealing the presence of communication between nodes, each edge is itself a 1-D heatmap of values characterizing this communication. In this section, we describe the procedure by which we compute the values for each edge and how we represent these values using edge coloring.

Latency is the amount of time it takes from the start of a request or send event until its processing starts or until the corresponding receive event ends. Depending on the layers at which events originate, latency is computed as follows:

- Between compute nodes and I/O (client) nodes: the difference between the start time of the first client event and the start time of the MPI-IO request; the "first client event" is the first I/O node event in chronological order which corresponds to the application I/O request. This is the I/O node's response to the compute node's request.
- Between I/O nodes and storage nodes: the difference between the end time of the ServerReceive event and the start time of the corresponding ClientSend event
- Time to write data to permanent storage: the difference between the end time of a server Write event and the start time of the same event. This metric is not currently present in our visualization, but adding it would be a very straightforward extension.

We compute latency values for each edge over the selected period of time. As previously mentioned, we need to match events across layers to achieve this. We do this by tracking the rank of the MPI process which initiated the I/O request and the ID of the request, which is unique per request per MPI rank. The coloring of the edges, shown in Figure 4, corresponds to a heat map of latency values. Latency is depicted by the position of the coloring on the edge. On each edge, the outermost point represents the highest latency value found in the data set, while the innermost point corresponds to the globally lowest latency values. We have chosen this ordering because it supports the perceptual association of larger values with the extra space available at the outer rim of a circle.

One problem that we discovered early on is the presence of negative latency values. These values occur due to slight clock desynchronization among the physical nodes running the application and I/O subsystem software. When tracing execution across numerous processors and over networks, it is impossible to keep the nodes' clocks fully synchronized without introducing frequent Barrier-type operations to resynchronize them. Introducing Barriers is undesirable because it would perturb the observed environment and negatively impact the performance of the applications we are analyzing. Instead, we adjusted for clock skew where we had MPI file operations in the traces that are known to be blocking operations, such as File Open. This operation acts as a Barrier, so it should release at approximately the same time on all the compute nodes. When we found discrepancies between the end times of File Open operations, we adjusted the timings of all application-level I/O events such that these discrepancies were eliminated. This greatly reduced the frequency of negative latency values between the compute processes and I/O nodes. However, we do not currently have a reliable method of adjusting this desynchronization across layers without introducing significant error in the data. We treat the negative values as simply the minimum observed latency and are thus still able to infer interesting and useful information regarding the relative latency among nodes and between different layers of the I/O system.

Because latency is heavily influenced by the amount of data being transferred through a particular link, we have also included the capability of visualizing **communication sizes** in our tool. Unfortunately, not all of our data sets contain buffer size information at the compute process level. When this information is present, it may be very helpful in explaining why certain processes experience higher or lower latency

levels than others, or it may indicate a performance problem when data sizes are disproportionately small in relation to the observed latency. In the graph visualization, buffer sizes are represented in the same manner as latency values are. Coloring at the outer edges of the circle represents the largest buffer sizes, while the center of the circle (edge destination point) corresponds to the minimum buffer size.

The color segments on the edges illustrate the number of times a certain latency value was encountered in the selected time period. We use two separate color maps for the two levels of edges, because using a single color map would result in different values mapping to the same color across the layers, which would be confusing. Grey means that no values were detected in the range. At the application–I/O node level, blue represents a low incidence of values, green represents the mid-range and yellow segments denote the latency values which were encountered the most. At the I/O node–storage node network level, the color progression is violet $\rightarrow$ red $\rightarrow$ orange. Color interpolation is performed on a logarithmic scale. We designed both these color gradients such that they have increasing luminance from top (for low values) to bottom (high values). Our intention was to avoid the confusion caused by color maps with random luminance variability [41].

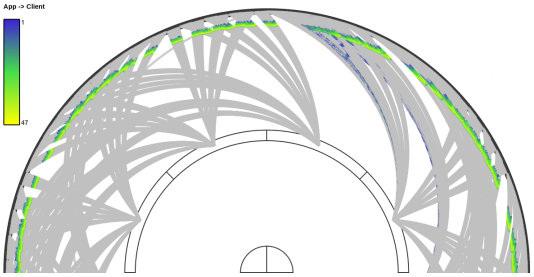
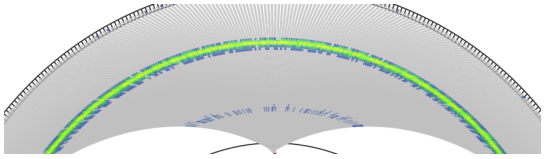
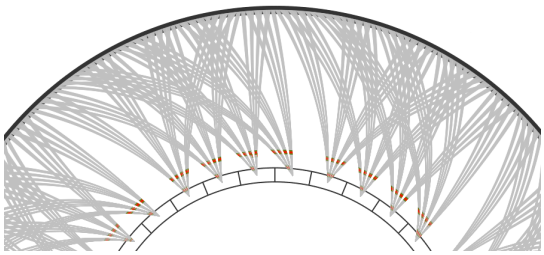


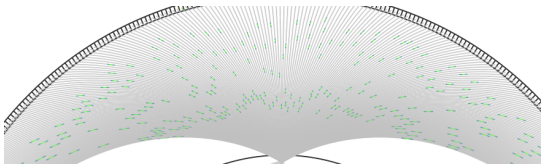
Fig. 4. Illustrating the color mapping: A section of the network activity graph between the compute nodes and I/O nodes from a FLASH trace on HDF5 with 2048 compute processes. The abundance of green and yellow coloration indicates that most links contain approximately the same number of latency value occurrences. The colored bands are distributed in a circular fashion, meaning that most communication across the layers occurred with around the same latency. The longer blue lines correspond to a small subset of compute nodes which experienced a wide range of latency values.



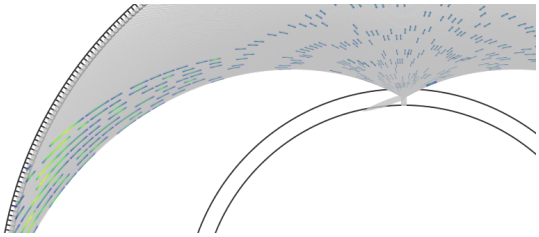
(a) **Circular alignment** of colored segments indicates that latency values are approximately constant, indicating a well-tuned and balanced I/O.



(b) A **striped pattern** occurs when the data is striped across various data storage servers, resulting in bands of nearby ranks accessing different I/O nodes. Depending on the network configuration, this kind of access pattern might not be beneficial.



(c) A **noisy pattern** results from a high variance in latency, as colored segments do not follow any particular alignment. This pattern is characteristic of initialization stages, but could indicate performance problems or imbalance if seen for an extended period of time during application execution.



(d) An **outlier behavior** occurs when a small number of compute nodes have higher latency than the rest, which is shown as a small number of colored segments at the very edge of the circle. This pattern could indicate severe imbalance, hardware failure, or special-purpose compute nodes that write much larger volumes of data.

Fig. 5. Common patterns in our latency visualization.

V. Results

To evaluate our method, we have collected a number of benchmark traces on the IBM Blue Gene/P systems Intrepid and Surveyor, which are housed at the Argonne National Laboratory, and visualized the latency recorded in the I/O network. In this section, we present the results of our case studies.

First, Figure 5 shows an overview of the latency patterns we most commonly saw in our study.

A. Chombo

The Chombo I/O benchmark is derived from Chombo, a parallel toolkit for adaptive solutions of partial differential equations [42]. Chombo is an interesting application because its I/O patterns are difficult for parallel file systems or I/O middleware to optimize. We performed our tests on Surveyor, with 512 compute processes, 2 I/O nodes and 2 storage nodes, using both the *in.r*222 and *in.r*444 example input files provided with the benchmark. The difference between these two input cases is that *in.r*444 writes a larger output file, approximately 18 GBytes in size.

In the r222 example, shown in Figure 6, latency values between the application and I/O node layers have a relatively wide distribution (shown as widely scattered blue regions), but a large number of values are concentrated in circular patterns. The different levels of these patterns actually correspond to

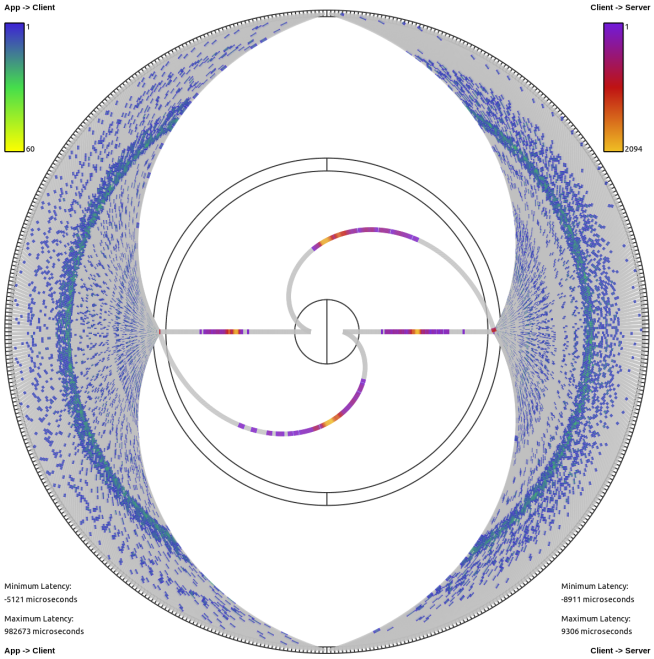


Fig. 6. Chombo r222 I/O latency over the entire execution. There is a wide latency distribution and an alternating pattern of high latency among some of the compute processes. These patterns correspond to the data write patterns observed in Figure 10.

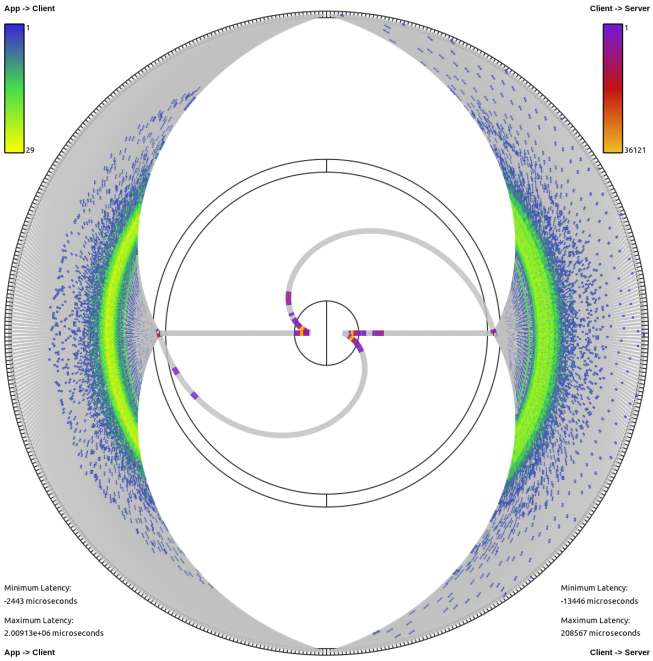


Fig. 7. Chombo r444 I/O latency over the entire execution. There is a more even distribution of relative latency values among the compute processes. The data write patterns for this example, shown in Figure 11, also indicate more consistency in this execution instance than the r222 case.

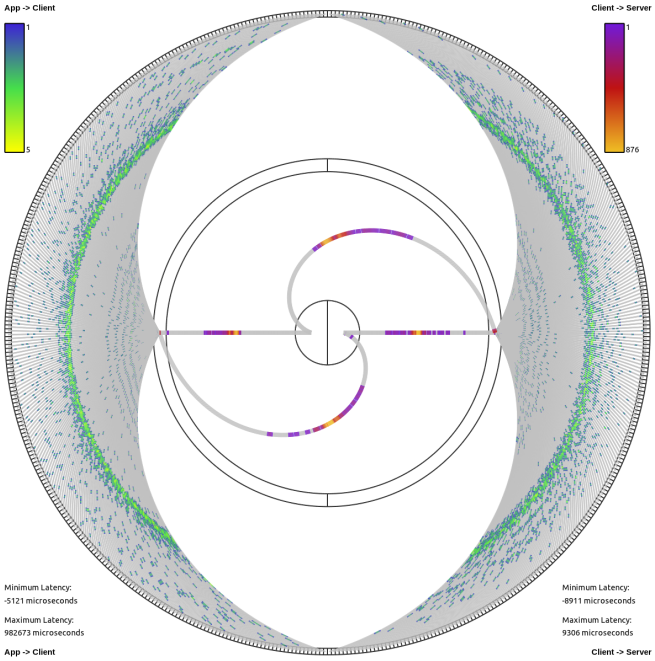


Fig. 8. Chombo r222 I/O latency—first half of the execution; all processes appear to experience approximately the same latency with respect to I/O node access. The striping latency pattern may be due to request queuing effects at the I/O node level.

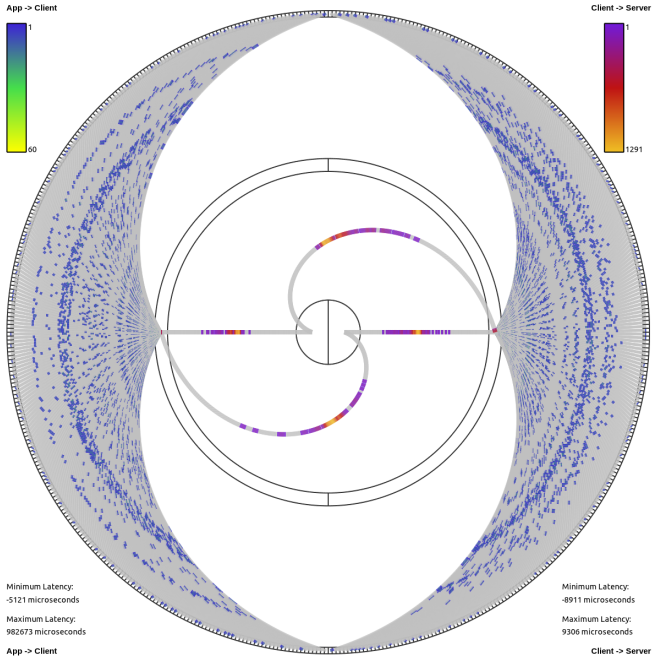


Fig. 9. Chombo r222 I/O latency—second half of the execution, showing a wider distribution of latency values and a number of outlying processes. These processes are responsible for writing the larger data buffers, which explains their higher latency values.

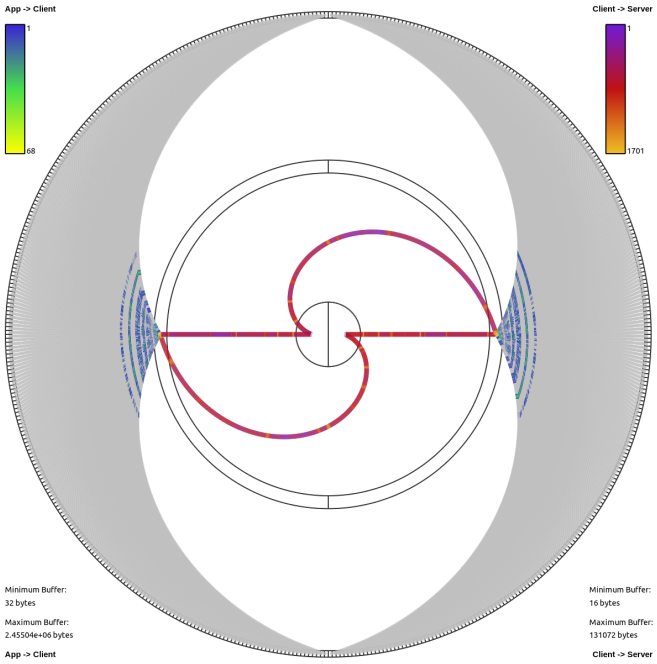


Fig. 10. Chombo r222 write buffer sizes throughout execution, showing a wide distribution of large and small data write requests. Colored segments indicate high to low buffer sizes, from the outside toward the inside of the circle and the scale is identical to the one in Figure 11. At the I/O node–storage node communication level, we notice that the buffers are divided up for storage, filling the entire range of sizes.

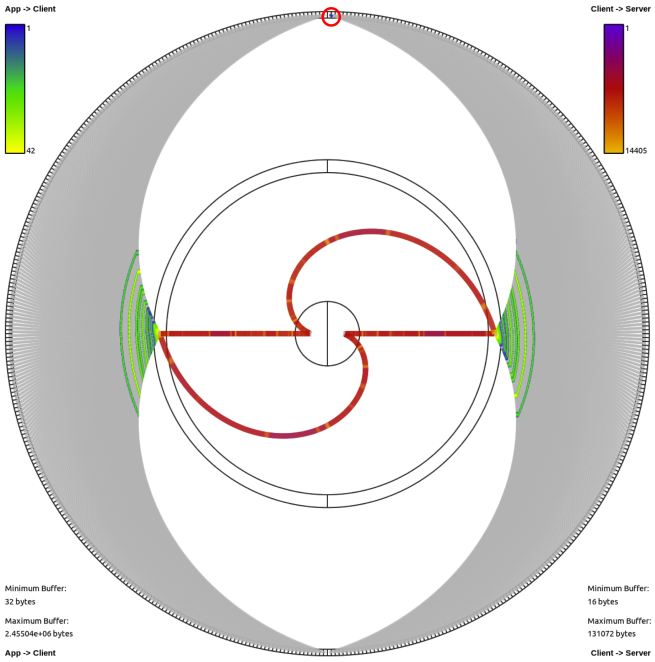


Fig. 11. Chombo r444 write buffer sizes throughout execution. This instance of the application mostly writes the same buffer sizes as the r222 case. Surprisingly, there is an instance of a very large buffer written by process 0 (highlighted by the red circle). If this buffer could be broken up into smaller pieces, it would likely improve the overall performance of the application.

different time periods in the execution. Between the I/O nodes and storage nodes we notice mostly mid-range latency values, but also with a wide distribution, as indicated by the relatively long length of the colored segments. The benchmark writes data using several buffer sizes, which correlates with the distribution of latency values.

By reducing the range of the temporal selection, we can see a change in access patterns between early (Figure 8) and late (Figure 9) in the application. The second half of the execution exhibits a wider distribution of values and the highest latency values recorded in this data set. The maximum latency, denoted by the blue coloring at the outermost edge of the circle, was observed at the end. This is possibly a result of the cleanup operations at the end of execution. There is also an alternating pattern of higher and lower latency values across the compute nodes, which indicates that some network links may have been saturated with requests and thus resulted in longer wait times for requests to be processed.

In the r444 case displayed in Figures 7 and 11, we notice that the majority of operations had similar latency values, as there is a thick circular band of yellow/green coloring in the visualization. This coloration indicates that there were more overlapping latency values observed in this trace. These values are also more centered around mid to lower range, potentially indicating better load balance or matching of resources to the task in this case. Processes 0–255 appear to have experienced overall higher latency, which may be due to the very large data request issues by process 0, which is highlighted in Figure 11.

By comparing these two separate executions of the same application, we notice that one of them appears to utilize I/O system resources more effectively. The r222 case, despite being shorter in duration, had a much wider distribution of latency values.

In Figure 10, we can see that there are some consistent buffer sizes being written, indicated by the concentric rings, but many of the larger data sizes appear to be randomly distributed among the compute processes. The r444 case has a much more concentrated pattern, both of latency values and of buffer sizes, indicating that it was better suited to utilize 2 I/O and 2 storage nodes. The single very large request issued by process 0 in this case is a performance bottleneck. As we see in Figure 7, it causes all the compute processes accessing the same I/O node as process 0 to experience higher request processing latency. If this data buffer could somehow be divided into smaller write requests, we would likely see a performance improvement.

B. *FLASH-IO*

FLASH-IO is the I/O kernel of FLASH, a multiphysics multiscale simulation code for general astrophysical hydrodynamics problems [39]. This kernel measures the I/O performance of the FLASH code by creating the primary data structures used in the simulation and then writing out example plot files. FLASH-IO writes data in three separate files: a checkpoint file, a plot file with centered data and a plot file with corner data. Our main FLASH-IO example is a trace collected on 2048

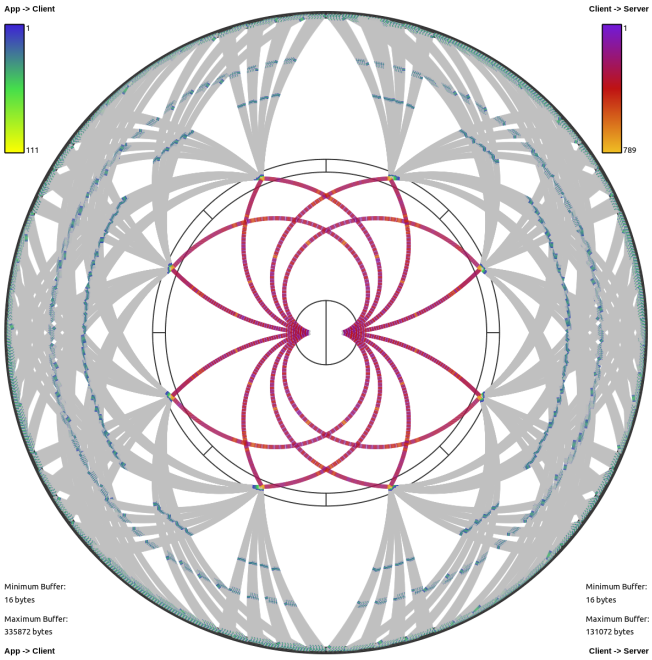


Fig. 12. FLASH-IO plot file write patterns (phases 2 and 3). The different buffer sizes are clearly visible. There is a set of large buffers being written by the application, as evidenced by the blue-green band at the periphery of the circle, two sets of mid-sized buffers, and a larger number of very small I/O operations. On the edges connecting the I/O nodes to the storage nodes, we can observe how these buffers are divided up for efficient storage. This also serves to balance the load for the I/O and storage nodes.

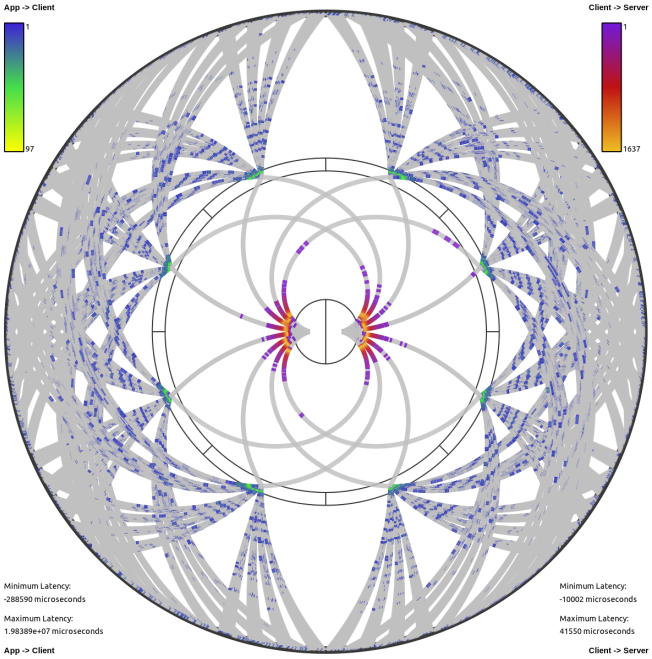


Fig. 13. FLASH-IO latency patterns during the writing of both plot files. At least five distinct rings are formed by the different latency values, more than the number of different buffer sizes written. This may be due to request queuing at the I/O node level. The outermost ring (highest latency) corresponds to the largest data buffers written. At the I/O node – storage node communication level, operation latency tends to be relatively low, as a result of breaking up large requests into smaller buffers for storage. However, one of the links displays a few instances of high latency, possibly indicating minor load imbalance.

compute processes, using 8 I/O nodes and 2 storage nodes to handle I/O requests (Figure 14). The data model used in this test is HDF5 with individual I/O requests (no collectives).

In the first phase, the benchmark is initialized and writes a checkpoint file. This accounts for one set of large-buffer write operations high to mid-range latency. The I/O system breaks up these large buffers for writing to permanent storage; the load on the I/O nodes and on the storage nodes appears to be balanced.

In the second and third phases of FLASH-IO (Figures 12 and 13), the two plot files are created. The first plot file appears to be sent to the storage nodes using larger buffer sizes than the second one. The maximum size buffers are still present, but in smaller numbers, possibly indicating further checkpointing that the application is performing to preserve partial results in case of a system failure. Although only four distinct buffer sizes are written at this stage, we can distinguish at least five circular latency bands in Figure 13. Additional latency may be caused by queuing of requests at the I/O node level. Each I/O node receives nearly simultaneous requests from 256 compute processes; if it is unable to process all these requests at once, it must leave some of the processes waiting for a brief period of time. Since all the latency rings appear to be even and encompass all the compute processes, we can conclude that the system is appropriately load-balanced and that all the processes have equal priority for the I/O nodes to process their

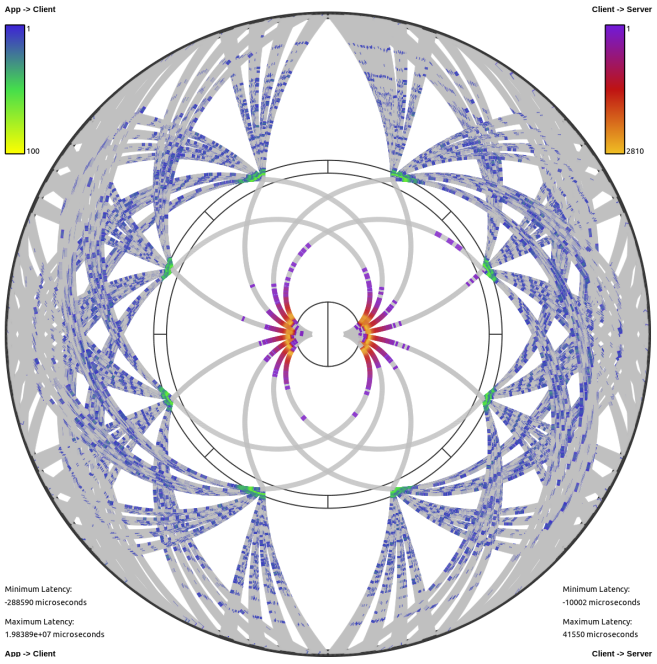


Fig. 14. Latency patterns in FLASH-IO for the entire execution. While values appear to be widely distributed between the compute nodes and I/O nodes, the green band around the edge of the circle indicates that there is a high incidence of maximum-latency events. As we will see shortly, this is because the benchmark writes numerous large buffers throughout its run.

data write requests. Another interesting observation is that the I/O nodes break up larger write requests, making them more efficient for the storage nodes to process and write. This is apparent in the wide distribution of message sizes between the I/O nodes and storage nodes, as well as by generally low latency values at this level.

VI. Conclusions and Discussion

Efficient file I/O is critical to the performance of large-scale scientific applications, and is expected to remain so given the continuously decreasing bytes/flops ratio. The scale of the contemporary HPC systems, coupled with their complex, multilayer hardware and software architecture, means that the performance measurement and analysis of HPC I/O systems is bound to be a formidable task. Even at relatively modest scale of a few thousand processes, jobs can generate millions of I/O events that take gigabytes of log space, with the resulting trace files beyond the human capability to sift through manually.

This work presented a novel radial visualization technique that lends itself well to the analysis of complex, multi-layer event traces. The technique was applied to the I/O traces of multiple application I/O kernels obtained on a Blue Gene/P system at a scale of several thousand processes, collected from three instrumented I/O layers: application processes, I/O nodes, and storage nodes. Radial graphs visualize the connections between these layers in a compact, easily comprehensible, not to mention attractive, fashion. The connections can be colored to show additional information about the I/O events, such as the latency or data payload size, providing an overview of both the raw values and their frequencies.

This approach can be used to analyze and compare existing I/O paradigms, revealing potential inefficiencies and bottlenecks. Our experiments with traces from production systems have revealed many interesting patterns; in addition to confirming an overall good tuning of the systems tested, we have also identified several cases that could result in inefficiencies and thus warrant further analysis. Techniques such as those presented here can reveal problems in applications and limitations in I/O systems that are not otherwise immediately apparent. We believe that our technique can be beneficial for the creators of applications and system software, as well as for the HPC system administrators, in the performance analysis, verification, and tuning of large-scale applications, I/O middleware, and HPC I/O systems—both existing and future.

VII. Future Work

To be a truly useful day-to-day tool for practitioners, our technique would need to be part of a more comprehensive tool set comprising multiple visualization techniques, advanced filtering and analysis, interactive drill-down techniques for in-depth studies, etc. While that goes beyond what a research project can deliver, we do intend to continue improving it.

Our method has proven to be effective at current scales, however, it is not without its limitations. At the anticipated Exascale counts of up to a million compute nodes and a billion compute threads, it will be necessary to perform pattern analysis and encode multiple nodes' communication into one edge.

The technique could be made more comprehensive by extending it to visualize additional layers, such as a high-level I/O library, I/O forwarding layer, or storage hardware. In the near future, we are looking to implement additional metrics, such as bandwidth per link or data throughput, into our visualization.

While our case studies were performed on IBM Blue Gene/P systems, our method is not limited to them, and it would be interesting to apply it to other HPC architectures and analyze their I/O patterns.

Acknowledgement

This work was supported by the National Science Foundation through NSF-0937928 and by the Office of Advanced Scientific Computer Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. Computing time on Intrepid was provided by a U.S. Department of Energy INCITE award and an ALCF Director's Discretionary Allocation.

REFERENCES

[1] W. Frings and M. Hennecke, "A system level view of petascale I/O on IBM Blue Gene/P," *Computer Science—Research and Development*, vol. 26, no. 3-4, pp. 275–283, Jun. 2011.

[2] A. Choudhary, W. keng Liao, K. Gao, A. Nisar, R. Ross, R. Thakur, and R. Latham, "Scalable i/o and analytics," *Journal of Physics: Conference Series*, vol. 180, no. 1, p. 012048, 2009. [Online]. Available: http://stacks.iop.org/1742-6596/180/i=1/a=012048

[3] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Proceedings of the 21st ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis (SC '09)*, Portland, OR, Nov. 2009.

[4] J. Fu, M. Min, R. Latham, and C. Carothers, "Parallel I/O performance for application-level checkpointing on the Blue Gene/P system," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, Sep. 2011, pp. 465–473.

[5] H. Yu and K.-L. Ma, "A study of I/O techniques for parallel visualization," *Journal of Parallel Computing*, vol. 31, no. 2, pp. 167–183, Feb 2005.

[6] W. Kendall, J. Huang, T. Peterka, R. Latham, and R. Ross, "Toward a general I/O layer for parallel-visualization applications," *IEEE Comput. Graph. Appl.*, vol. 31, no. 6, pp. 6–10, Nov. 2011.

[7] P. C. Roth, "Characterizing the I/O behavior of scientific applications on the Cray XT," in *Proceedings of the 2nd International Workshop on Petascale Data Storage (PDSW '07)*, Reno, NV, Nov. 2007, pp. 50–55.

[8] D. J. Kerbyson and P. W. Jones, "A performance model of the parallel ocean program," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 3, pp. 261–276, Aug. 2005.

[9] LANL-Trace, http://institute.lanl.gov/data/software/#lanl-trace.

[10] S. Seelam, I.-H. Chung, D.-Y. Hong, H.-F. Wen, and H. Yu, "Early experiences in application level I/O tracing on Blue Gene systems," in *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS '08)*, Miami, FL, Apr. 2008.

[11] B. J. N. Wylie, M. Geimer, B. Mohr, D. Böhme, Z. Szebenyi, and F. Wolf, "Large-scale performance analysis of Sweep3D with the Scalasca toolset," *Parallel Processing Letters*, vol. 20, no. 4, pp. 397–414, 2010.

[12] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, 2006.

[13] Sandia National Laboratories' Red Storm I/O traces, http://www.cs.sandia.gov/Scalable_IO/SNL_Trace_Data/index.html.

[14] N. Nakka, A. Choudhary, R. Klundt, M. Weston, and L. Ward, "Detailed analysis of I/O traces of large scale applications," in *HiPC, International Conference on High Performance Computing*, Dec. 2009.

[15] The SYSIO library, http://sourceforge.net/projects/libsysio.

[16] S. J. Kim, Y. Zhang, S. W. Son, R. Prabhakar, M. Kandemir, C. Patrick, W.-k. Liao, and A. Choudhary, "Automated tracing of I/O stack," in *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface (EuroMPI '10)*. Stuttgart, Germany: Springer-Verlag, 2010, pp. 72–81.

[17] B. Mohr and F. Wolf, "KOJAK—A tool set for automatic performance analysis of parallel programs," in *Euro-Par 2003 Parallel Processing*, ser. Lecture Notes in Computer Science, H. Kosch, L. Boszormenyi, and H. Hellwagner, Eds. Springer Berlin / Heidelberg, 2003, vol. 2790, pp. 1301–1304.

[18] F. Wolf and B. Mohr, "Automatic performance analysis of MPI applications based on event traces," in *Proceedings from the 6th International Euro-Par Conference on Parallel Processing (Euro-Par '00)*. London, UK: Springer-Verlag, 2000, pp. 123–132.

[19] ——, "Automatic performance analysis of hybrid MPI/OpenMP applications," in *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on*, Feb. 2003, pp. 13–22.

[20] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr, "Scalable parallel trace-based performance analysis," in *In Proc. 13th European PVM/MPI Conference*. Springer, 2006, pp. 303–312.

[21] A. Knüpfer, B. Voigt, W. E. Nagel, and H. Mix, "Visualization of repetitive patterns in event traces," in *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, ser. PARA'06, 2007, pp. 430–439.

[22] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp, "From trace generation to visualization: A performance framework for distributed parallel systems," in *Proc. of ACM/IEEE Supercomputing (SC00)*, November 2000.

[23] E. Karrels and E. Lusk, "Performance analysis of MPI programs," in *Proc. of the Workshop on Environments and Tools for Parallel Scientific Computing*, J. Dongarra and B. Tourancheau, Eds. SIAM Publications, 1994, pp. 195–200.

[24] V. Herrarte and E. Lusk, "Studying parallel program behavior with `upshot`," Argonne National Laboratory, Tech. Rep. ANL–91/15, 1991.

[25] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with Jumpshot," *Int. J. High Perform. Comput. Appl.*, vol. 13, no. 3, pp. 277–288, Aug. 1999.

[26] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir performance analysis toolset," in *Tools for High Performance Computing*, M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds. Springer, Berlin, pp. 139–155.

[27] C. Muelder, F. Gygi, and K.-L. Ma, "Visual analysis of inter-process communication for large-scale parallel computing," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1129–1136, October 2009.

[28] C. Muelder, C. Sigovan, K.-L. Ma, J. Cope, S. Lang, K. Iskra, P. Beckman, and R. Ross, "Visual analysis of I/O system behavior for high-end computing," in *Proceedings of the 3rd International Workshop on Large-Scale System and Application Performance (LSAP '11)*, 2011, pp. 19–26.

[29] T. Ludwig, S. Krempel, M. Kuhn, J. Kunkel, and C. Lohse, "Analysis of the MPI-IO optimization levels with the PIOViz Jumpshot enhancement," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, F. Cappello, T. Herault, and J. Dongarra, Eds. Springer, Berlin, 2007, vol. 4757, pp. 213–222.

[30] H. Jagode, A. Knupfer, J. Dongarra, M. Jurenz, M. S. Mueller, and W. E. Nagel, "Trace-based performance analysis for the petascale simulation code flash," 2010.

[31] PVFS2: Parallel Virual File System, version 2, www.pvfs.org.

[32] P. Braam, "The Lustre storage architecture," http://www.lustre.org/docs/lustre.pdf, 2004.

[33] F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters," in *First Conference on File and Storage Technologies (FAST)*, 2002.

[34] N. Ali, P. Carns, K. Iskra, D. Kempe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable I/O forwarding framework for high-performance computing systems," in *IEEE International Conference on Cluster Computing (Cluster 2009)*, 2009.

[35] K. Ohta, D. Kempe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa, "Optimization techniques at the I/O forwarding layer," in *IEEE International Conference on Cluster Computing (Cluster 2010)*, 2010.

[36] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman, "ZOID: I/O-forwarding infrastructure for petascale architectures," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, 2008, pp. 153–162.

[37] "mpi-tile-io," http://www.mcs.anl.gov/research/projects/pio-benchmark/.

[38] The IOR benchmark, http://sourceforge.net/projects/ior-sio/.

[39] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo, "FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes," *The Astrophysical Journal Supplement Series*, vol. 131, no. 1, p. 273.

[40] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, no. 5, pp. 741–748, Sept.-Oct. 2006.

[41] D. Borland and R. Taylor, "Rainbow color map (still) considered harmful," *Computer Graphics and Applications, IEEE*, vol. 27, no. 2, pp. 14–17, Mar.–Apr. 2007.

[42] "Chombo—Infrastructure for adaptive mesh refinement," https://commons.lbl.gov/display/chombo.